AD-A236 655

(2)

USAISEC

US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300

U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES

ARMY NONPROGRAMMER SYSTEM FOR WORKING ENCYCLOPEDIA REQUESTS PHASE IIIA REPORT (ASQB-GT-91-007)

JANUARY 1991

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800

This research was performed by Honeywell Federal Systems, contract number DAKF11-88-C-0024, for the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), the RDTE organization of the U.S. Army Information Systems Engineering Command (USAISEC). The report discusses a set of data management tools and distributed query processing modules developed and/or enhanced during a 9 month effort, the third phase of a four phase effort. Requests to view a demonstration of the prototype may be made by contacting MAJ(P) David S. Stevens at 404/894-3110. This research report is not to be construed as an official Army position, unless so designated by other authorized documents. Material included herein is approved for public release, distribution unlimited. Not protected by copyright laws.

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

s/ Glenn Racine

Glenn Racine, Chief
Computer and Information
Systems Division

s/ John R. Mitchell

John R. Mitchell
Director
AIRMICS

Accesion For	
NTIS CRA&I	✓
DTIC TAB	
Unannounced	
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

Table of Contents

Table of Contents (Concluded)

List of Figures

Section 1
Introduction

This report covers the Phase IIIA activities of the ANSWER program from December 16, 1989 to September 30, 1990.

In Phases I and II of the ANSWER program, we emphasized the development of data administration tools to aid in the process of creating and maintaining a heterogeneous distributed system. The overall ANSWER architecture as envisioned as the end of the program is shown in Figure 1-1. For review, deliverables for Phases I and II are highlighted in the figure and are described below.

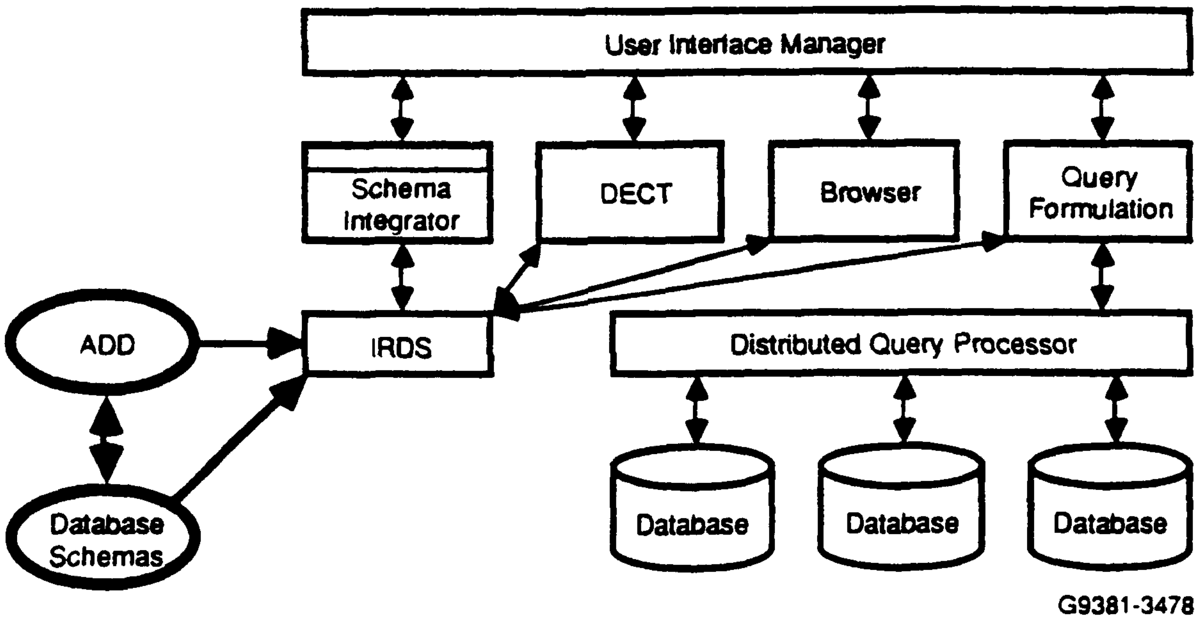

Figure 1-1. ANSWER Chart

The schema integrator creates integrated schemas from individual schemas and data administrator assertions about individual schema relationships. The integrated schemas can be used as the basis of distributed query processing by providing a global definition of available data and static mappings to local systems. The schema integrator reads and writes schemas to and from an IRDS repository, an ANSI and FIPS standard for data definition repositories. All ANSWER tools will eventually share information through the IRDS repository.

The browser and schema editing facilities provide a graphical browser for entity-category-relationship (ECR) schemas and for information about standard data elements. The browser/editor also reads information from and writes information to IRDS. The editor supports the graphical creation of schemas and the inspection of associated standard data element information. The browser also supports the browsing of integrated schema information and associated individual schema information.

The Army Data Dictionary (ADD) is an implementation of the ADD currently under development at the Data Management Directorate (DMD) at Ft. Belvoir. The schema used in the current ANSWER prototype is a subset of the schema used for the ADD by DMD. Extensions may be made to the ANSWER ADD to store additional information.

The Data Element Creation Tool (DECT) is a tool designed to support the creation of standard data elements. The tool is intended to be used at the installation level as an aid to developing new standard data element names. It supports the creation of a syntactically correct name (in accordance with AR 25-9) and also supports the semantic analysis of that name to determine if any other closely related names have already been proposed. The analysis is based on an internal representation of the semantic structure of the data element name and may be related to the use of that name in a schema.

The User Interface Manager is a multi-window graphical user interface that supports control integration of multiple tools. Each tool is represented as a state transition diagram, which is managed by the interface manager. The state transition diagram controls the points at which the user may suspend execution of one tool and invoke execution of a different tool. The user interface manager also controls the tool interfaces, managing input and output for each tool as well as window managing functions and graphical display functions required by the tools.

Further information about these tools may be found in the Phase I and II final reports.

1.1 Phase IIIA

Phase IIIA addressed the query formulation tools associated with ANSWER. In this phase we developed a query formulation tool and experimented with different query processing algorithms for a distributed query processing environment. Figure 1-2 shows the Phase IIIA deliverables.

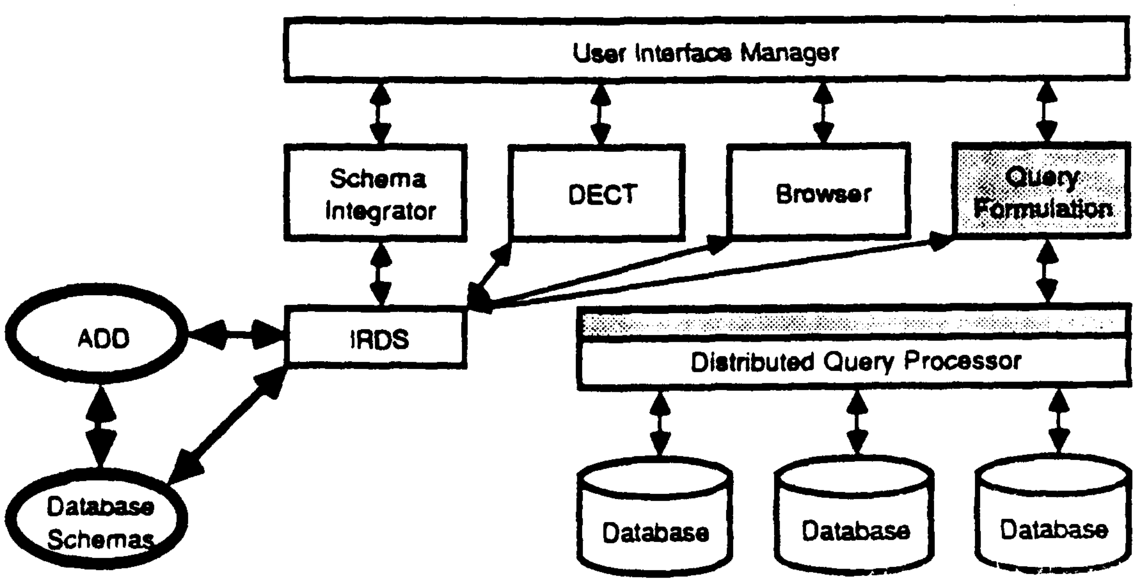

Figure 1-2. ANSWER Chart

The specific tasks of Phase IIIA are described below. This report discusses each of these tasks in greater detail in the following sections.

1.1.1 Query Formulator

The query formulator task addresses the design and development of a query formulation aid for end users that supports the formulation of syntactically correct SQL queries. In Phase IIIA we addressed the design and initial implementation of query formulation. We addressed the problem of formulating queries directly against individual or integrated schemas, and we addressed the problem of formulating queries using standard data elements instead of schemas. The query formulation tool produces queries executable by an SQL processor for queries stated against individual schemas.

1.1.2 Query Processing Algorithms

For queries stated against integrated schemas or stated using standard data elements, additional processing is necessary to obtain an executable SQL query. In Phase IIIA we investigated several alternative query processing algorithms. We have implemented two algorithms for query processing that address the issue of query decomposition, execution and result composition for queries against the integrated schema. One algorithm is designed for use with standard data element queries and the other is designed for use with standard integrated schema queries. Section 3 discusses the two algorithms, their strengths and weaknesses and makes recommendations for query processing algorithms appropriate for distributed query processing in ANSWER.

1.1.3 Distributed Query Processing

As part of Phase IIIA we also needed to identify existing commercially available distributed query processing capabilities. These capabilities are to be used as a target for a design of the integration of ANSWER tools with a distributed query processing environment.

Section 2
Query Formulator

The ANSWER query formulator is a tool designed to aid end users in creating well formed SQL queries. The query formulator produces queries executable by an SQL processor. The tool provides both graphical and textual means to create and edit queries.

In constructing the query formulator we had several goals:

- A "modeless" environment, where users can freely move between text and graphic creation of query terms;
- Incremental query development, where a user may test out individual select clauses before composing a complex embedded query;
- Alternative views of query formulation, including a relational view and a standard data element view (following AR 25-9).

This section will discuss the architecture of the query formulator including the query editor, the condition expression editor, and standard data element query formulation.

The Query Formulator (QF) provides a syntax-directed, window-based tool for defining SQL queries. A QF user may enter and edit queries by pointing with the mouse at items on the screen, by selecting options from context-sensitive pop-up menus, or by typing information to the editor. Thus, QF supports multiple modes of interaction in the query definition process. To focus the editing process, individual conditions and expressions in a complex query can be edited independently. The entire query is always visible so that, while editing a particular component of the query, the user can view that component within its context. Furthermore, through the use of separate "query buffers", the user can work on several independent queries simultaneously. With this mechanism subqueries can be developed separately and then combined through straightforward linking operations. This encourages an incremental strategy in defining complex queries.

Queries can be defined in either of two modes—relation mode and standard data element (SDE) mode. Queries defined in SDE mode are automatically converted into relation mode when query execution is requested so that they can be executed as standard relational SQL queries. The user can also request that any query in SDE mode be converted into relation mode without executing the query. The resulting query (or queries) can then be edited further in relation mode.

2.1 Invoking the Query Formulator

There are two ways to invoke QF: (1) as an independent tool or (2) from the grapher window.

In the first approach, QF is invoked through a pull-down menu available from the top level menu bar in the ANSWER system. An initialized query buffer is created, and the user can define a query in this buffer, as shown in Figure 2-1.

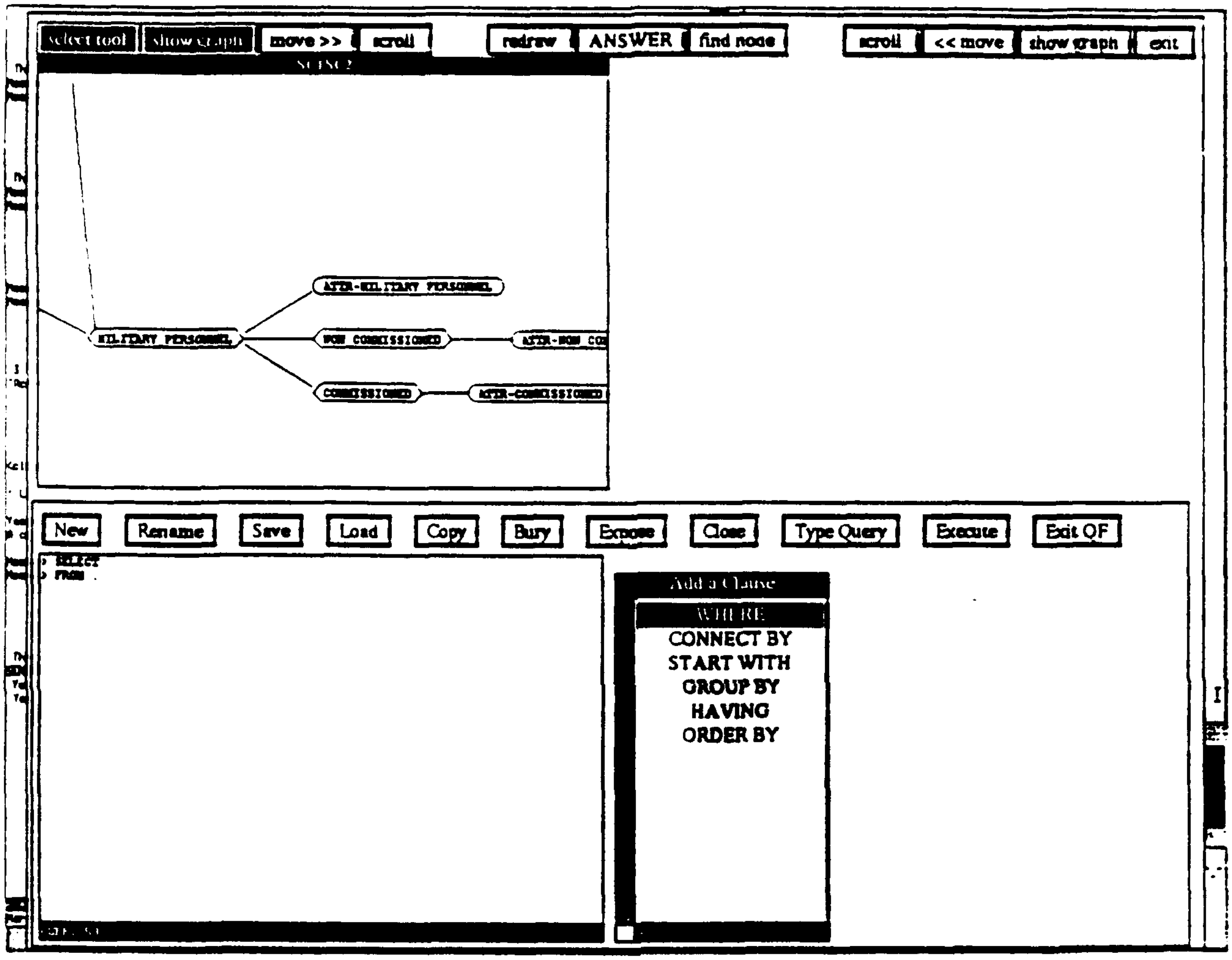

Figure 2-1. Query Formulator Initial Screen

Under the second approach, the user indicates through a menu command attached to some node in a grapher window that she wishes to define a query. This node is taken as a starting point in the query specification. QF will be started and an initial buffer will be created. If the node corresponds to a relation or a prime term, then the relation or prime term will be included in the FROM clause of the new query. If the node corresponds to an attribute or a standard data element, then the attribute of data element will be included in the SELECT clause. The user may also request a new buffer from within QF. An initialized buffer will be created and exposed, and the user may then edit this new query.

If QF has previously been invoked in the user session, an additional invocation will not cause a new QF to be created. Rather, the existing QF will be activated (it is inactive), and a new buffer will be created containing any initial information specified through the invocation.

Note that when QF is invoked from a relation or attribute grapher node, the buffer created will automatically be in relation mode. When it is invoked from a grapher prime term or standard data element node, the buffer will automatically be in SDE mode.

2.2 Query Editor

The Query Editor supports the editing of SQL queries. The term "clause" denotes any top-level component of an SQL query. A query always contains a SELECT clause and a FROM clause. It may also contain WHERE, CONNECT BY, START WITH, GROUP BY, HAVING, UNION, INTERSECT, MINUS, and ORDER BY clauses. The term "select statement" denotes a group of clauses headed by a SELECT and FROM clause. A select statement contains no more than one of each of the above clauses. Thus, a subquery is not considered part of the containing select statement. The subquery constitutes an independent select statement distinct from, but related to, the select statement that contains it.

Each query in QF is presented to the user in a query buffer. (Figure 2-1 shows a sample initial layout for a query buffer.) In the left-hand portion of the QF window is the query display window, which shows the current version of the query. If there are multiple select statements in the query, all of them are shown, but only one of them can be edited at any one time. We call this select statement the "current select statement." The current select statement is indicated by a surrounding rectangle. The select statement for any sub- or superquery can be made into the current select statement simply by clicking on that select statement. A select statement may have any number of immediate subqueries but no more than one immediate superquery. Both select statements and entire queries may be duplicated. The query display window can be scrolled vertically. Each query (and buffer) has a name associated with it. The name of the currently displayed query is shown at the bottom of the query display window in inverse video.

The right-hand portion of the QF window changes depending upon the state of the query formulation process. In the initial state this portion contains a menu of the clauses that can appear in a select statement (see Figure 2-1). These menu items have two options; (1) create clause with editor or (2) create clause from type-in.

When the first option is selected, the right-hand section of the QF window is replaced by a Condition/Expression Editor (see the subsection on this below) in which the user may define an SQL condition or expression specifying the indicated clause. This editor is subordinate to the current select statement. Once the user has defined the condition or expression, the clause is added to the current select statement, the Condition/Expression Editor is removed, and the clause menu reappears. When the user requests the second option, a type-in window appears over the clause menu. The user may type any appropriate clause, and the result will be added to the current select statement. Each clause may be used no more than once in a single select statement. After they are used, clauses are shaded out.

At the top of the QF window is a menu of commands for manipulating query buffers, which we will now describe . The first one is NEW. This command creates a new initialized buffer and exposes it. The new buffer contains only a single select statement (i.e., there are no sub-

or superqueries). This select statement consists of an empty SELECT clause and an empty FROM clause. A name is automatically generated for the buffer.

The second command is RENAME. It asks the user for a new name to be associated with the buffer and select statement. This new name is associated with the select statement until the name is changed again or the select statement is discarded.

Next is the SAVE command, which stores the current query to a file. The user is prompted for a file name where the query will be stored. (A reasonable default should be supplied.) The SAVE command stores the current form of the query. Since the query might be in an incomplete state, the saved representation must be capable of expressing these states. That is, the stored definition might not be a valid SQL query. In particular it is possible to have multiple unconnected pieces of single conditions or expressions. The SAVE command should also store the name of the query. The LOAD command retrieves a query from a file. It must read the output produced by SAVE and reconstruct the internal representation of the stored query. A buffer is created to hold the query.

The COPY command is actually two commands. When the user selects this command, a pop-up menu appears that has the following two choices:

- Copy current select statement,
- Copy entire query.

If the user selects the first option, the current select statement is duplicated in another buffer. A name for the new buffer is automatically generated (e.g., Copy_of_<name>). The second option works in much the same way except that the entire query is duplicated in a new buffer. A new name is generated for the buffer.

The next two commands control which buffer is seen (and edited) by the user. That is, they control which buffer is current. The first of these is BURY. Query buffers are conceptually stacked on top of each other. The BURY command takes the top buffer (the visible one) and moves it to the bottom of the stack. Whatever buffer was second on the stack will now be the current buffer. The EXPOSE command gives the user a menu of all the buffers in the buffer stack from top to bottom. The user may select any buffer. That buffer is moved from its current position to the top of the stack and becomes the current buffer. The CLOSE command removes the current buffer from the buffer stack without saving the associated query. The user is asked to verify this step.

The TYPE QUERY command allows the user to type an SQL query into a window appearing over the clause menu. A new buffer is created to hold the query. Note that once the user has defined a new query through type-in, that query may be edited using the means described below.

The EXECUTE command executes the current query. The results appear over the clause menu. Finally, the TRANSLATE command converts a query in SDE mode into a query in relation mode. One or more new buffers are created to hold the new query.

2.2.1 Editing Scenario

Now we are ready to consider in more detail how queries are edited. We have already discussed the way in which new clauses are added. To review, the SELECT and FROM clauses are automatically created for each query buffer. The other clauses of a select statement can be added through the clause menu. We first consider the editing of SELECT and FROM clauses. In the following, many of the editing commands are available in pop-up menus, activated by a mouse click on the right button. The left and middle buttons are generally unused. (At a later date, special commands could be attached to these buttons to speed up the execution of those commands.) The user can select a pop-up menu option to proceed, or can move the mouse off of the pop-up menu, in which case the original command is canceled. We should also note that all of these editing commands apply only to the current select statement, not to its sub- and superqueries.

The FROM clause contains a list of relation names or prime terms. Clicking right on the word FROM produces a menu with two options—DEFINE BY POINTING, and DEFINE BY TYPING. With the first command the user indicates a relation or prime term by clicking on a corresponding object in the graphics window. With the second, the user is asked to type the name of a relation or prime term. Once a relation is in the FROM clause it too is mouse-sensitive. The right-button menu contains three commands: DECLARE LINK, DECLARE ALIAS, and DELETE ITEM. The declared links and aliases must be typed by the user. The new link or alias replaces any existing link or alias. The DELETE ITEM command simply removes the indicated item from the FROM list.

The SELECT clause contains a list of expressions. The menu on the word SELECT allows the user to add a new expression at the end of the list, either through an expression editor (see next subsection) or through type-in. It also contains commands to declare the ALL/DISTINCT keyword for the SELECT clause. The default is ALL. The right-button menu on SELECT items contains commands to edit the item in an expression editor, to replace the item through type-in, to delete the item, and to modify the order of the select items. The order modification commands are SWAP, PUT BEFORE, and PUT AFTER. The GROUP BY clause has similar behavior. The difference is that it lacks the ALL/DISTINCT commands on the clause word GROUP BY. Similarly, the ORDER BY clause has no ALL/DISTINCT commands and it has ASC/DESC commands associated with each ORDER BY item.

The other clauses have only two options: modify in editor, or retype. These options are available through the right-button menu on the clause word. Since there can only be one item associated with each of these clauses (e.g., the WHERE clause allows only a single condition), these operations always replace the previous value rather than adding a new one to the end of the list as for SELECT, ORDER BY, and GROUP BY.

While a condition/expression is being edited, the clause or expression is highlighted in the query buffer to indicate that it is the one being edited.

2.3 Condition/Expression Editor

Now we turn our attention to the Condition and Expression Editors. These are really just two separate modes of the same editor. In expression mode, the editor expects the user to define an expression that can be used in a SELECT, GROUP BY, or ORDER BY clause. It restricts the operators available to those that can be used in such an expression. In condition mode, this editor expects the user to define a boolean condition that can be used as a WHERE, CONNECT BY, START WITH, or HAVING clause. Since such a condition can have any kind of expression as a component, all of the operators from expression mode are available here as well. In addition, the boolean operators AND, OR, and NOT, and relational operators such as `=', `>', IN, and EXISTS are accessible. In reality, a variety of more restricted modes are used by the system to enforce type checking. These will be covered below. In this subsection we will for the most part discuss the Condition and Expression Editors as a single entity, drawing distinctions between the two major modes when necessary. We call this combined editor the Condition/Expression Editor (C/E Editor).

The C/E Editor is drawn in the right portion of the QF window (see Figure 2-2). The editor is itself divided into three windows: the combination window, the term window, and a menu pad. The left portion of the C/E editor is occupied by the combination window. This window is used for building complex conditions and expressions from simple ones. The simple expressions consist of terms and numbers. Terms are shown in the term window, which occupies the lower right portion of the C/E editor. Terms include: attribute names, qualified attribute names, SDEs, subqueries, and strings. Subqueries are not actually displayed in the term window. Instead, QF automatically generates a name for each select statement. That name is printed in the term window. There is a VIEW SUBQUERY command attached to these names, which allows the user to see the select statement referred to by a particular name. Associated with each term in the term window is an identifier (a letter), which is used to denote the term in the combination window. Both the combination and term windows can be scrolled vertically.

The third component of the C/E editor is the menu pad. It appears above the term window and contains three menu options: NEW TREE, NEW TERM, and DONE. The DONE command is used to indicate that the user has finished editing the condition or expression. The resulting value is entered into the appropriate spot in the query, and the C/E editor is deactivated.

New terms may be entered into the term window by selecting the NEW TERM command in the menu pad. Clicking right on this button brings up a menu with the following options: SELECT GRAPHICAL OBJECT, TYPE TERM, CREATE NEW SUBQUERY, and USE EXISTING SUBQUERY. The user chooses one of the above commands and then supplies the term. For the SELECT GRAPHICAL OBJECT option, the user can click on any appropriate node in one of the grapher windows. The selected term is added to the term window (an identifier is automatically generated for this new term). The CREATE NEW SUBQUERY option creates a new select statement as a subquery of the current select statement. An automatically generated name for the subquery is added to the term window with a new identifier. This new subquery is immediately displayed in its incomplete form in the query buffer (see the comments on displaying incomplete queries). When the user selects USE EXISTING SUBQUERY, she is given a menu of queries known to QF. A copy of the selected query is

made, and this copy becomes a subquery of the current select statement. The subquery is immediately displayed within the proper context in the query window. When TYPE TERM is chosen, the user is given a type-in window. The entered term is added to the term window with a new identifier. New terms may also be created through certain commands available in the combination window. These will be described below.

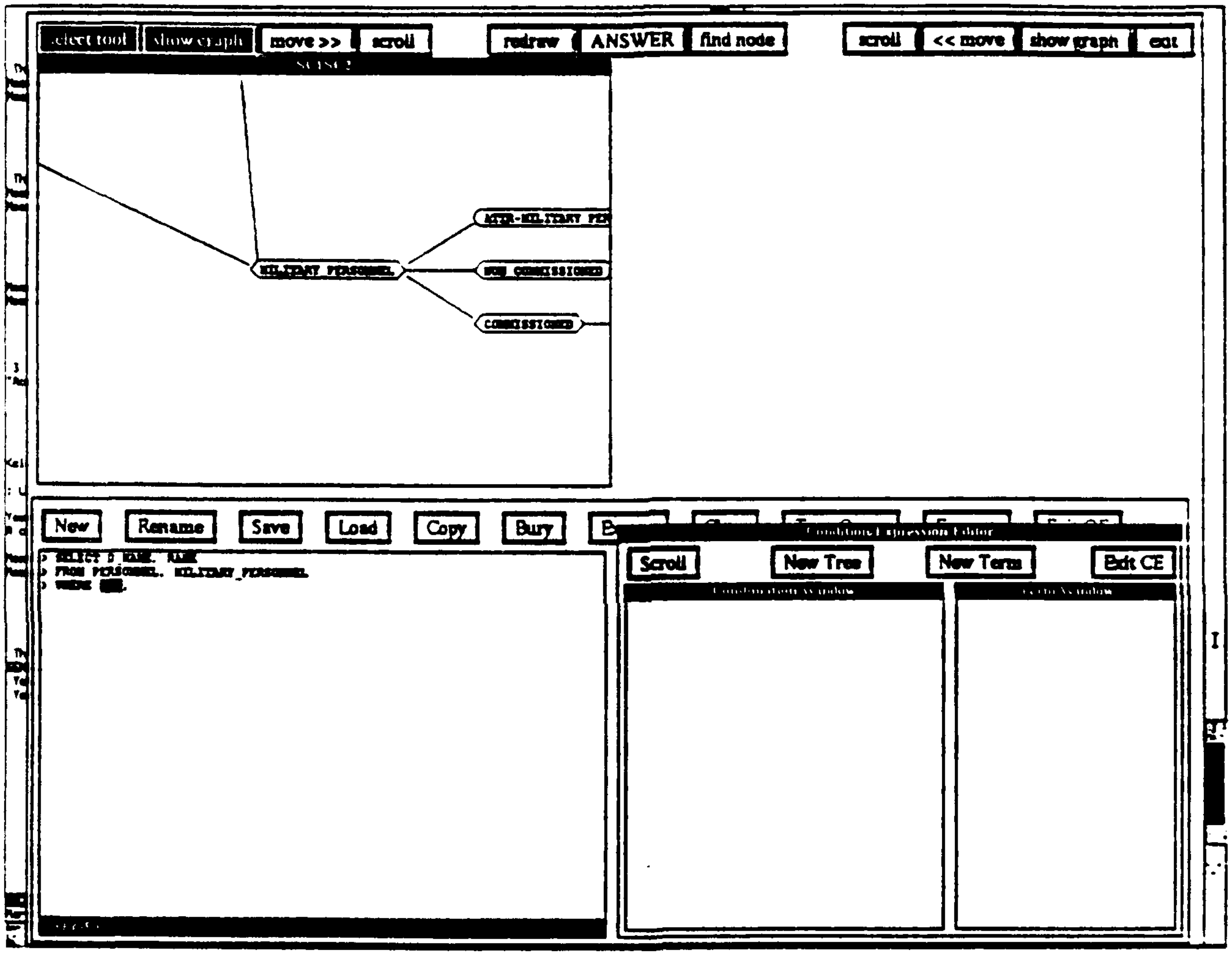

Figure 2-2. Condition/Expression Editor Invoked

The contents of the combination window is a set of trees drawn vertically from the root down to the leaves. These trees reflect the structure of conditions and expressions in a query. Each tree and subtree represents the application of some operator to its arguments. Thus, each nonleaf node is an SQL operator. Each leaf node in a complete condition/expression is either a number or a term identifier. The arguments are taken from left to right in building SQL expressions from these trees. While trees are being built, their leaf nodes may be unspecified. Such a leaf node is called "open." All other tree nodes are "closed." Place holders are displayed in the combination window for open leaf nodes. The holders have names appropriate to the corresponding arguments of the parent node operator. Thus, if the first argument of an IN operator is unspecified, the leftmost child of the IN node might have the name "attr." If the second argument is unspecified, the rightmost child might have the name "rel" (see Figure 2-1). Open leaf nodes are graphically

distinguished in some way (e.g., an italic font or underscoring). To be executable, a condition/expression should have no open leaf nodes. During the process of defining a condition/expression, there may be multiple trees. Each of these trees represents an independent condition or expression. Various commands are available to link trees together to form one complete condition/expression. An executable condition/expression can have only one tree. For flexibility, QF should be able to store conditions and expressions with open leaf nodes and multiple trees. These partially specified conditions and expressions can then be completed in a subsequent editing session.

New trees can be created by selecting the NEW TREE menu option from the menu pad. The options presented to the the user will depend on the mode of the C/E editor. If a condition is being edited, every type of operator will be available. If, on the other hand, an expression is being edited, some operators will be unavailable. The precise list of available operators depends on the type of expression being defined (e.g., SELECT clause versus GROUP BY clause). The specifics will be presented below when we discuss modes in greater detail.

2.4 Operations on Open Leaf Nodes

The basic operations on open leaf nodes are SHOW SQL DOCUMENTATION, CONNECT TO, DEFINE FROM MENU, TYPE, and PASTE. The CONNECT TO operation fills in the open node with a previously defined tree or term. The user simply selects this command on the open node and then selects CONNECT FROM on the root of a tree or the identifier of a term. The indicated item is spliced into the tree at the point of the open node. When the TYPE command is issued the user is allowed to type a term, condition, or expression which is then analyzed and spliced into the tree at the point where the open node was. Figures 2-3 and 2-4 show a CONNECT TO operation.

The DEFINE FROM MENU command presents a menu of things that could fill the open node. The content of the menu depends on how the open node is used in the tree. If the node should be filled by a condition (e.g., the node is a child of AND), then only condition operators are presented. If the node should be filled by a numeric expression (e.g., the node is a child of '+'), the menu allows the user to select numeric-valued functions, numbers, numeric-valued attributes, etc. Again, this is heavily dependent on the mode system to be described later.

The PASTE and COPY (available on closed nodes) commands work together. QF has a scratch pad in which any expression or condition can be stored. Whenever a COPY command is issued on any closed tree node (leaf or nonleaf), the expression or condition denoted by that tree or subtree is placed in the scratch pad. The previous contents of the scratch pad are discarded. A subsequent PASTE operation on any open node will cause the contents of the scratch pad to be inserted for that node. This process differs from a CONNECT TO in that a copy of the expression is made rather than directly hooking the expression into place. Furthermore, a CONNECT TO can only be done within a single C/E editor, but a COPY/PASTE operation sequence can copy information from one C/E Editor to another.

The GET SQL DOCUMENTATION command displays information about the parent operator and the purpose of the argument defined by the open node.

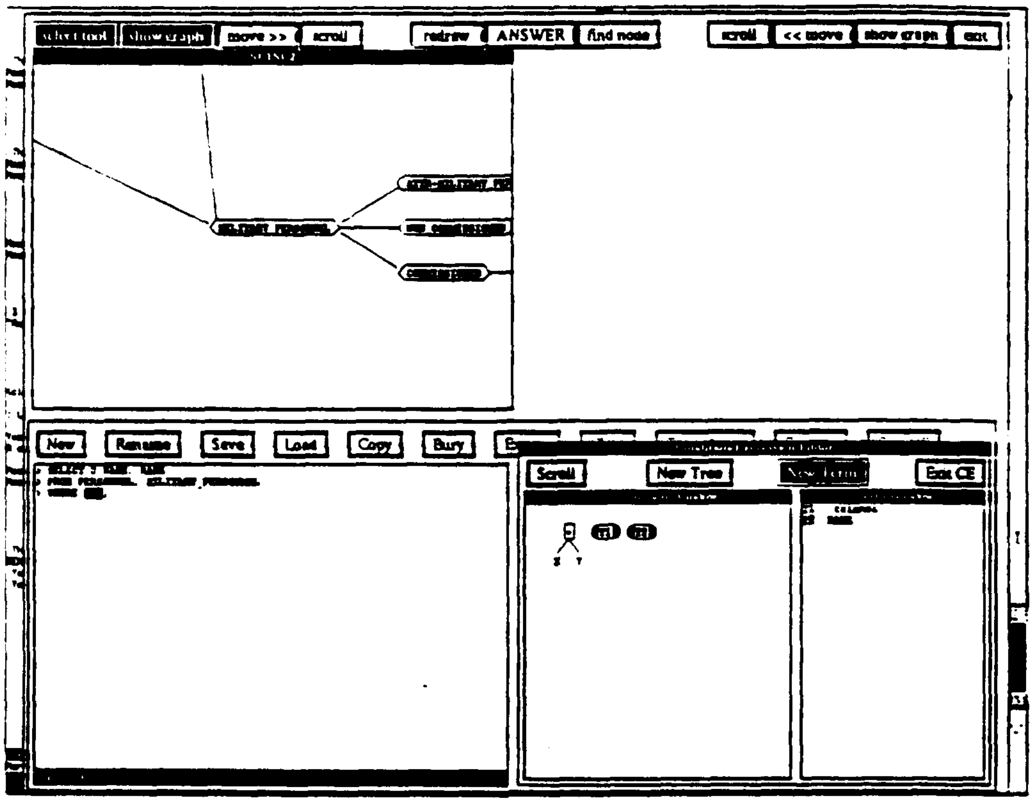

Figure 2-3. Before CONNECT TO Operation in C/E Editor

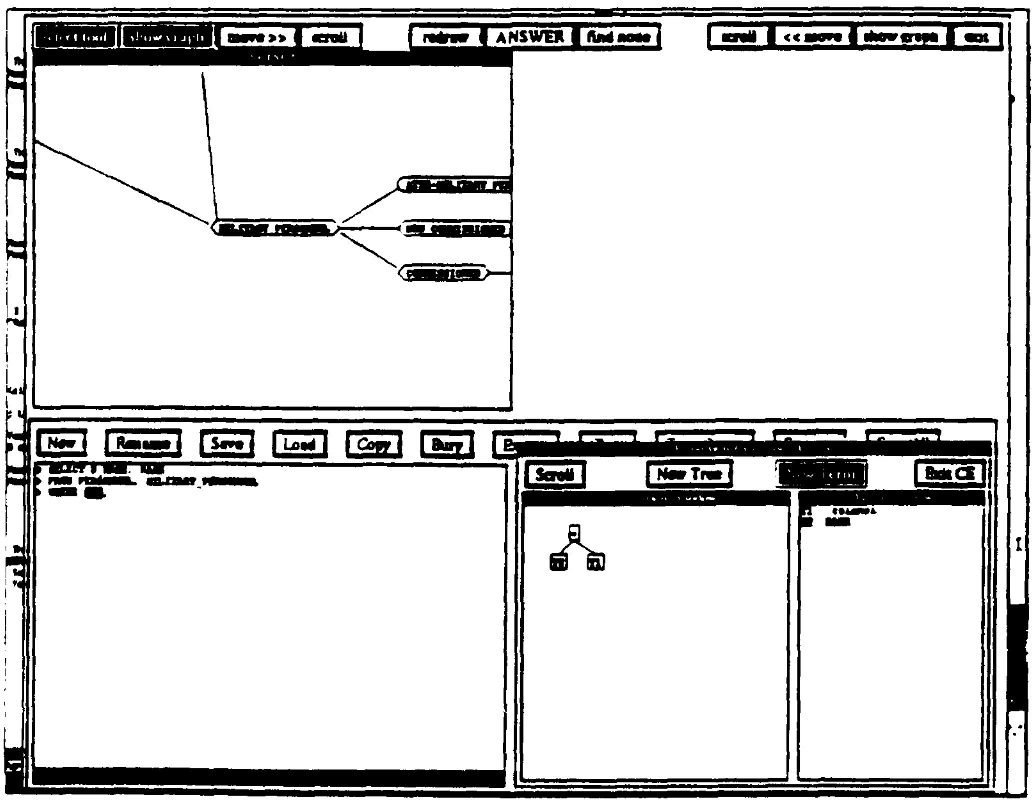

Figure 2-4. After CONNECT TO Operation in C/E Editor

2.5 Operations on Nonleaf Nodes

The basic operations on nonleaf nodes are COPY, DETACH TREE, DELETE TREE, GET SQL DOCUMENTATION, and ADD CHILD and DELETE CHILD (in the correct context). COPY has already been discussed. DETACH TREE is roughly the inverse of CONNECT TO. It takes the subtree whose root is the node from which the DETACH TREE command is executed and turns it into an independent tree. The root node reverts to an open leaf node. DELETE TREE removes the indicated node and all its children from the combination window. The node reverts to an open leaf node. GET SQL DOCUMENTATION displays a description of the operator and its arguments in a notification box.

Finally, the ADD CHILD command is used to expand the number of arguments for an operator that takes an arbitrary number of them. (This does not include operators with optional arguments. These arguments are included automatically along with some indication of their default value. See the discussion of special arguments below.) The operators that take an arbitrary number of arguments are AND, OR, VALUE-LIST, `+', `*', DECODE, GREATEST, and LEAST. All of these operators except DECODE have one argument that can be replicated. DECODE has a pair of arguments representing a search condition and a return value, and these are replicated together. The ADD CHILD command simply adds one or more new child nodes at the right of the string of replicatable arguments. For example, if we perform an ADD CHILD operation on a LEAST node with children nodes for the terms A and B, a new child will be added to the right of the B node. If we perform the same operation on a DECODE node with children expr, search1, return1, search2, return2, default, then two new children, search3 and return3, are added between return2 and default.

The final nonleaf node operation is DELETE. As noted above this command is available only in certain situations. Some SQL operators take an arbitrary number of arguments. For example, the LEAST operator takes two or more arguments. The nodes for these operators have special commands for adding children (see the subsection on nonleaf nodes). The DELETE command is used to remove these extra children when they are no longer wanted. In that case the child node is removed, and the number of arguments is reduced by one.

2.6 Operations on Closed Leaf Nodes

Closed leaf nodes are treated as a special case of closed nodes. They have only three commands: DELETE-NODE, COPY and DETACH-NODE. These commands have the same effect as the analogous commands for nonleaf nodes.

Section 3
Query Processing Algorithms

Under Phase IIIA we have developed two alternative query processing algorithms:

- Term substitution algorithm,
- Approximate query processing.

The two algorithms highlight different issues in the area of query processing against heterogeneous databases. The term substitution algorithm is similar to algorithms developed in the context of distributed homogeneous databases (e.g., [CERI83]. The approximate query processing algorithm is based on work done by [BUDW89], which addresses techniques for processing queries against categories which are not strictly defined as part of the integrated schema of local database schemas managed by a heterogeneous database management system (HDBMS).

The investigation of approximate query processing was prompted by the need to consider issues associated with processing queries stated using standard data elements where there is not a simple one-to-one mapping between SDEs and elements of a given integrated schema. The term substitution algorithm is currently used with queries stated directly against the integrated schema.

Distributed query processing requires at a minimum some approach to the following issues:

1. Decomposition of a global query into locally executable queries,

2. Management of execution of the local queries including optimization and transaction management,

3. Recomposition of the local query results into a final result.

At this phase in the project we have addressed issues 1 and 3;. we have not addressed the problems in 2. The two algorithms we implemented under Phase IIIA will be presented and discussed in the next subsection. The results of our investigations and the next step in developing practical query processing algorithms for the heterogeneous environment will be discussed in the conclusions of this subsection.

3.1 Term Substitution Algorithm

Our term substitution algorithm is based on an algorithm developed by [CHUN90], which was developed for use in an HDBMS environment. That algorithm did not correctly address the treatment of queries stated against generalization hierarchies such as the one shown in Figure 3-1 below. The problem is that queries stated against such hierarchies must support the recomposition of results using a operations such as union and outerjoin [DAYA84].

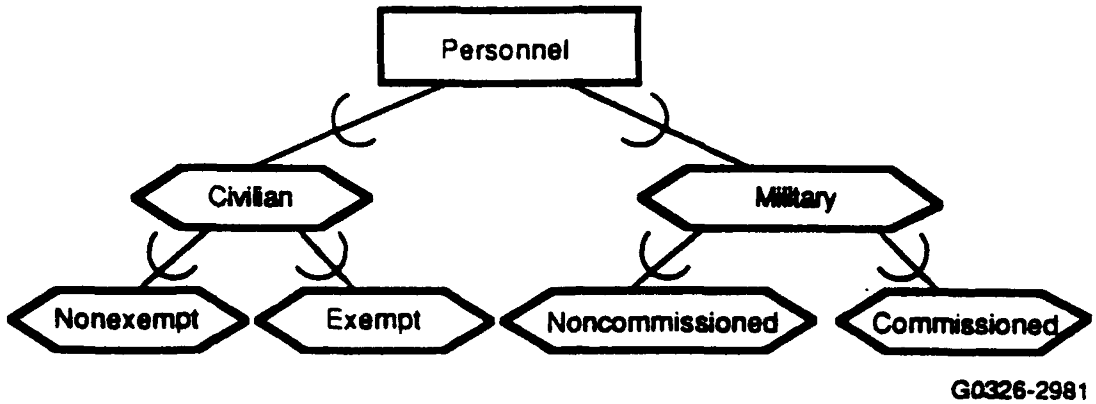

Figure 3-1. Generalization Hierarchy in an ECR Schema

We modified this algorithm to perform unions of local query results as well as joins of local query results to correctly recompose results of queries at the global level. Additional work is required to adequately incorporate the use of outerjoin and other operations.

The term substitution algorithm currently in use is given below.

3.1.1 Term Substitution Strategy

1. Map the query to individual schemas using mapping rules which state translations of integrated schema terms to local schema terms
2. If the query maps completely to more than one schema (the current version of the algorithm is written to support two schemas only), then UNION the results of each local query execution
3. If the query does not map completely to either local schema but there are global join conditions stated in the FROM clause of the global query, then execute the partial queries and JOIN the results using the global join conditions.
4. If the query maps completely to one schema but does not map completely to the other schema, execute the query against the first individual schema only.

The mapping procedure used by the term substitution algorithm follow [CHUN90]. The mapping procedure translates a global query into local queries executable on different databases. The local queries must be saved into temporary relations and recomposed into a final result following a result query. The steps are presented in more detail below.

1. Assign target attributes from the global select clause and global aggregate clause to the result query.
2. For each relation R in the query, substitute R with the local relations R(I) that R maps to and associate a set of query attributes A(I)with R(I), where A(I) is the intersection of all attributes defined for the relation R(I) with the target attributes from the global query.

3. Identify any global join terms in the query and add those terms to the WHERE clause of the result query.

4. For each relation R(i), associate additional attributes with A(i). The additional attributes are the global join terms, which are also attributes of R(i).

5. Create local queries by assigning all R(i) of the local schema to the FROM clause. Assign all attribute sets A(i) of the R(i) to the SELECT clause. Assign all local join terms to the WHERE clause. Execute each query and store the results in a local relation L(i).

Once the local queries have been executed, the final result is composed following the term substitution strategy outline above. This is the point where our algorithm differs from the [CHUN90] algorithm.

An example query decomposition is shown below in Figure 3-2. The case shown is one in which the query maps completely to each of two local schemas. The local schemas and integrated global schema are shown in Figures 3-3, 3-4 and 3-5.

GLOBAL QUERY:

Select D_Item_No, D_MName

From E_Materiel_Item

Where D_Item_No > 500

LOCAL QUERY 1:

SELECT SC1_MATERIAL_ITEM.ITEM_NO,

SC1_MATERIAL_ITEM.MNAME

FROM SC1_MATERIAL_ITEM

WHERE (SC1_MATERIAL_ITEM.ITEM_NO > 500)

LOCAL QUERY 2:

SELECT SC2_MATERIAL.M_ID, SC2_MATERIAL.MNAME

FROM SC2_MATERIAL

WHERE (SC2_MATERIAL.M_ID > 500)

Figure 3-2. Sample Query Decomposition *(continued on page 3-4)*

```
FINAL RESULT QUERY:
SELECT                         SC1_MATERIAL_ITEM.ITEM_NO,
SC1_MATERIAL_ITEM.MNAME
FROM SC1_MATERIAL_ITEM
WHERE ( SC1_MATERIAL_ITEM.ITEM_NO > 500 )
UNION
SELECT SC2_MATERIAL.M_ID, SC2_MATERIAL.MNAME
FROM SC2_MATERIAL
WHERE ( SC2_MATERIAL.M_ID > 500 )
```

Figure 3-2. Sample Query Decomposition (Concluded)

The mapping of each query is based on a set of mapping rules which state the relationships between the local schemas and the global schemas. Currently those rules are stored in local files. Eventually the rules should be stored in the IRDS repository along with the schemas. The rules are given in Appendix A.

3.2 Approximate Query Processing

The approximate query processing algorithm is based on work done at the University of Pennsylvania by [BUDW89]. They develop an algorithm for query processing that produces a set of definite answers, a set of possible answers and a set of anomalous tuples for a query. The determination of definite, possible or anomalous answers is based on a set of rules which defines the query terms. For example, assume a query such as:

```
Select name, performance_rating
From hourly_employee
```

For this query, the algorithm would require a set of rules that defines hourly_employee in terms of the schema elements. In this case, the term hourly_employee may be defined by rules similar to the following:

```
SC1:secretary -> hourly_employee

SC1:carpenter -> hourly_employee

SC2:non-commissioned -> hourly_employee

hourly_employee -> IS:Personnel
```

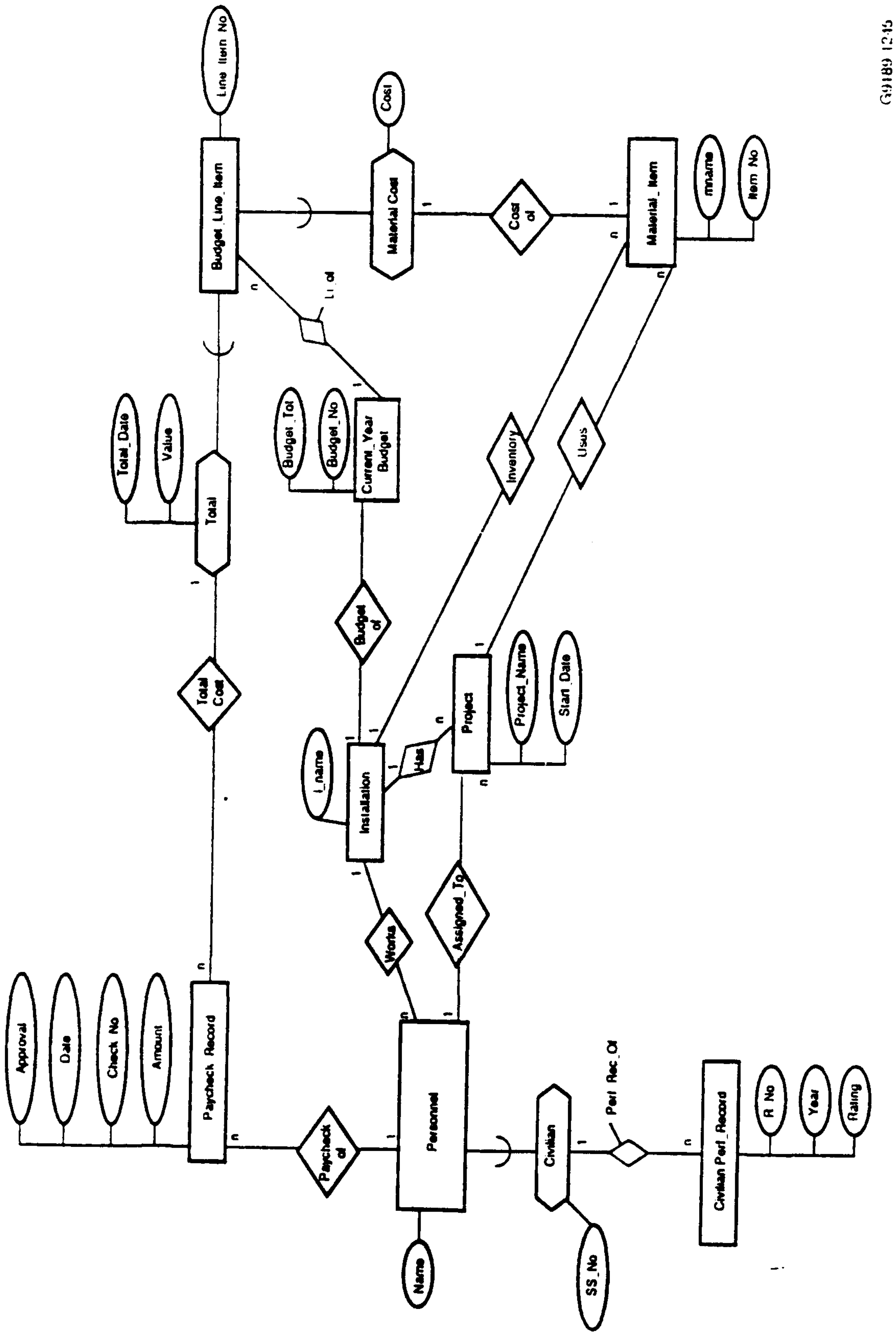

Figure 3-3. Local Schema 1

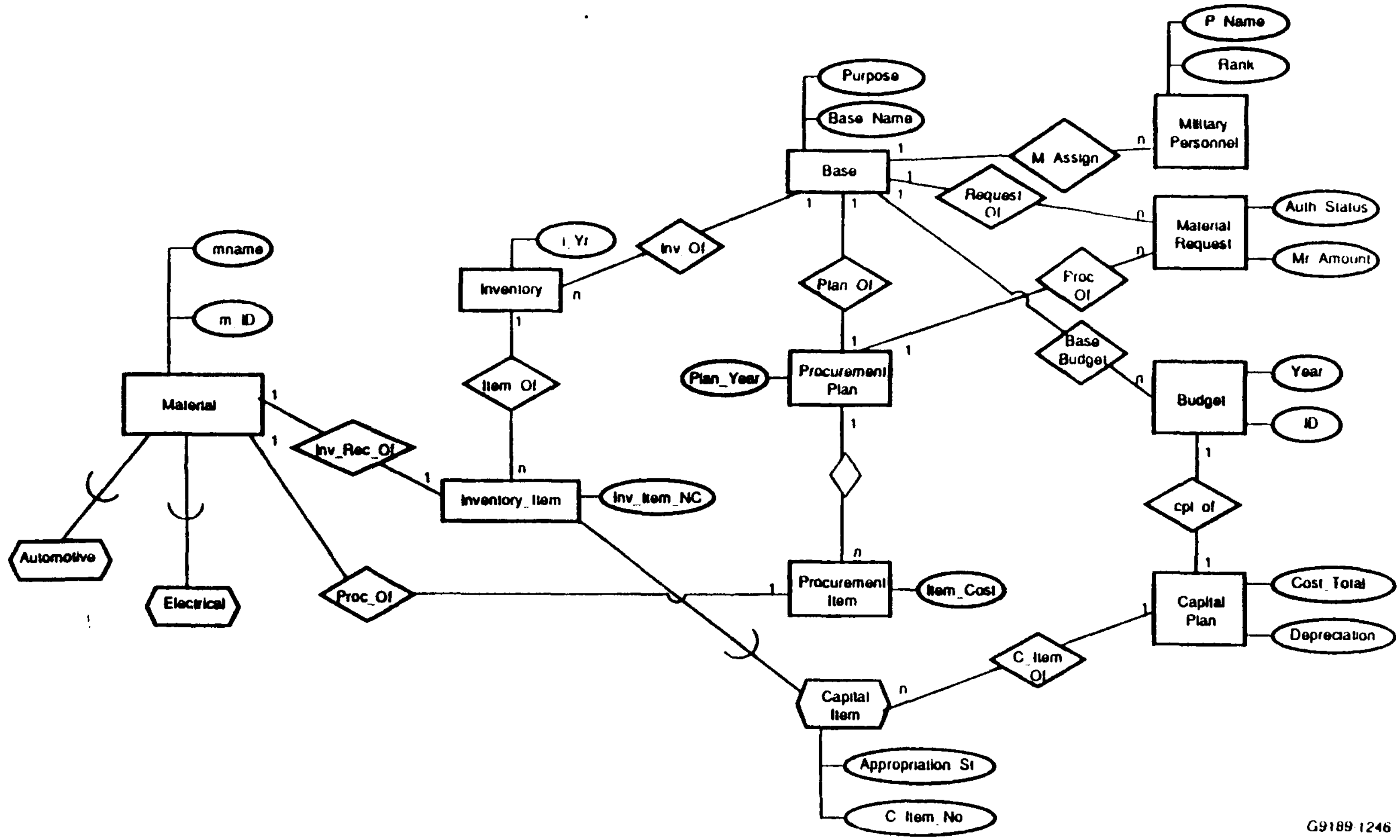

Figure 3-4. Local Schema 2

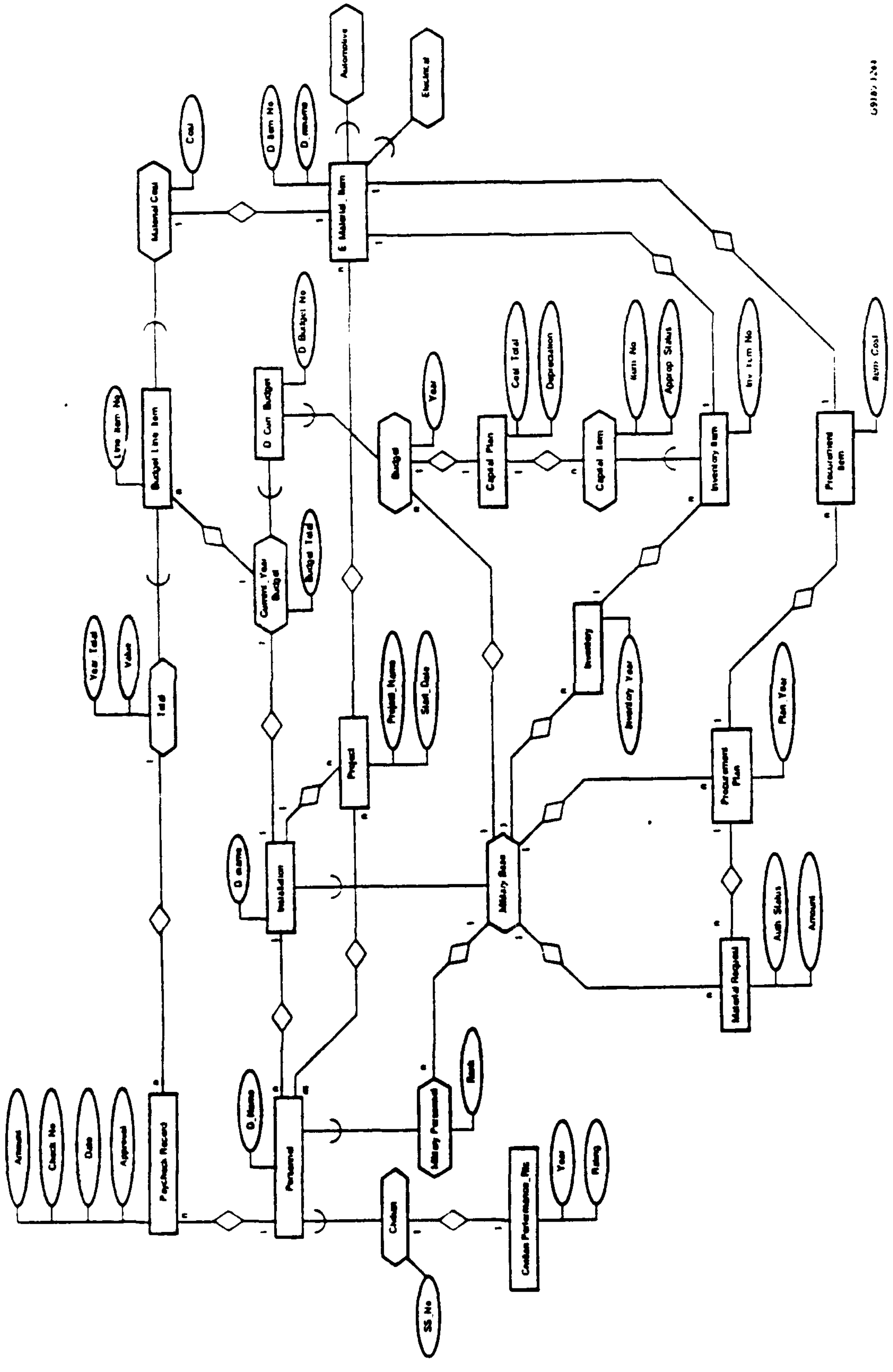

Figure 3-5. Integrated Schema

The rules describe 'is-a' relationships between the term hourly_employee and the elements of the schema. The basic idea behind the algorithm is to use the rules relevant to a particular query to guide the process of constructing two different sets of tuples—essentially all tuples that may be answers to the query (MAYBE) and all tuples that are definitely answers to the query (DEFINITE). The MAYBE answers are tuples that occur in relations named on the right-hand side of the rules. In this case, the space of all possible hourly employees is defined by the tuples in IS:Personnel, SC1:Civilian and SC2:military. The DEFINITE answers are defined by tuples that occur on the left-hand side of the rules, in this case SC1:secretary, SC1:carpenter, and SC2: Non-commissioned.

The basic ideas behind the algorithm can be given as follows:

1. Select rules relevant to a query.

2. Create a set of DEFINITE answers by computing an APPROXIMATE UNION of schema terms on the left-hand side of the rules.

3. Create a set of MAYBE answers by computing the JOIN of all schema terms named in the right-hand side of the relevant rules, RHS, and compute RHS MINUS DEFINITE.

4. Improve the DEFINITE answers with additional information from the RHS tables by joining tuples with identical key values.

5. Identify ANOMALOUS tuples as tuples which are in the DEFINITE answers but which are not in the MAYBE answers.

The algorithm is an effective approach to computing answers for queries about elements that are not strictly part of the defined schema structure. In the example presented here, we treated the term hourly_employee as a term that could be used to refer to a category of individuals that is not explicitly defined in the database schemas. Hourly_employees are defined to be subtypes of Civilian personnel that include secretaries and carpenters but not engineers. To extend the problem to apply to both sample schemas we choose to treat a subcategory of military personnel, non-commissioned, as hourly employees for sake of developing the argument. In this situation, hourly_employees is a category that introduces a cross classification of existing schema information. It is not present as an explicit category in either schema. The advantage of the approximate query processing algorithm is that it can compute the tuple that satisfies the definition of this cross classification. This is important for distributed query processing because it would not always be necessary to anticipate all possible cross classifications that some new user queries might require for processing. It is difficult to anticipate all such categories and without some approach to approximate query processing such queries cannot be satisfied.

We investigated the algorithm as an approach to the problem of supporting queries phrased using standard data elements. The scenario we envisioned is one satisfying the following characteristics:

1. The standard data elements may have been developed independently of any schemas for the databases that the SDEs describe.

2. The queries will be stated in terms of standard data elements but must be translated to queries executable against existing database schemas.

3. Some data administrator has defined the standard data elements in terms of the target schema elements (using rules similar to the ones presented above.)

4. The standard data elements, because they were developed independently of existing schemas may not have a clean mapping to existing schema categories. This may happen because the Prime-Word + Modifier construction may allow the SDE developer to create categories for which instances exist in the database, but for which there exists no isomorphic category. Personnel Hourly employee is an example of such a case. The instances satisfying that description exist in the databases but there is no explicit category that neatly maps to the existing database schemas.

Given such a situation there are two alternatives:

- Redesign existing schemas so that they conform to the category distinctions maintained by registered SDEs.

- Develop approaches to query processing that will allow queries phrased in terms of SDEs to be executed against existing schemas.

The first of these alternatives, redesign existing schemas, has serious practical problems in terms of cost for schema redesign and maintenance. The second alternative, developing new approaches to query processing, has several benefits. It may provide adequate support for querying existing databases through the alternative views provided by SDEs. In addition, even if queries are phrased using only existing schema terms, similar situations may arise in the distributed processing of queries against heterogeneous independent databases, that is, there must be some kind of support for approximate query processing whether or not the SDE view of data is maintained.

The approximate algorithm we investigated for Phase IIIA shows that:

- Approximate query processing is feasible.

- Approximate query processing is applicable to the problem of supporting SDE queries against existing schemas.

There are limitations to the existing algorithm that we plan to address in Phase IIIB. The existing algorithm is well defined for queries within individual-type hierarchies. That means that all rules used to answer a given query must reference the same keys in relations being manipulated. The applicability of the algorithm is not as clear when queries extend beyond single type hierarchies. That means any queries involving joins of individuals from IS:Personnel of any of the subtypes of IS: Personnel with individuals from IS: Installation or other parts of the schema cannot be appropriately satisfied. We are planning to investigate the applicability of a newly defined relational algebra that allows for approximate answers in the same sense as [BUDW89] but which may have broader applicability. It is clear from our work in Phase IIIA that some approach to approximate query processing is required independently of any approach requiring the use of SDEs to formulate queries. As a side benefit, any approach to approximate query processing that is generally applicable should work as well to support processing of SDE queries.

Section 4
Distributed Query Processing Platform

Phase IIIA addressed the issue of what distributed query processing platform should be used for the ANSWER tool set. The configuration for using ANSWER tools within a distributed query processing environment is shown in Figure 4-1.

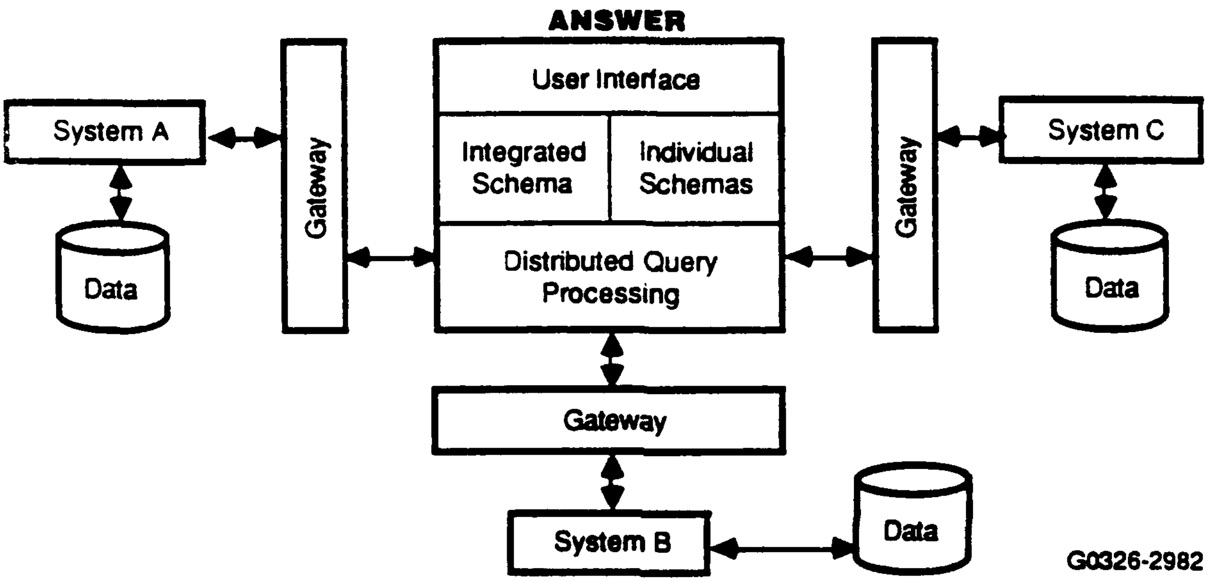

Figure 4-1. ANSWER/Distributed Query Processing

ANSWER tools reside on a central workstation running client applications that access the data servers running on possible other platforms. For Phase IIIB, the distributed query processing will be the AIRMICS testbed running SQL*NET. ANSWER will supply the end user tools, data administration tools and query processing algorithms that will enable a system to access data through the integrated and individual schema views maintained by ANSWER.

For example, ANSWER may be used as an intermediary to finance, supply and accounting systems to manage the exchange of information about contracts, vendors, inventory, and funds. In particular, ANSWER could be used as an intermediary to manage requests for data from Army systems such as SAACONS, FESS and STANFINS.

Section 5
PHASE IIIB and PHASE IV Plans

Phase IIIA addressed the issues associated with end user query formulation tools and initial algorithms for distributed query decomposition and execution. Phase IIIB will look at enhanced algorithms for query decomposition, investigating the applicability of an extended relational algebra to use in formulating distributed heterogeneous queries and will investigate associated integrity and security issues. The tasks for Phase IIIB and Phase IV are as follows.

5.1 Phase IIIB

October 1, 1990 to February 28, 1991

This phase is a six-month phase, which will address issues associated with query decomposition in a heterogeneous environment.

5.1.1 Task IIIB.1—Implement Browser/UIMS Enhancements

Investigate the feasibility of porting the ANSWER tool set to an appropriate hardware platform for execution with the identified distributed query processing system.

5.1.2 Task IIIB.2—Query Formulation Implementation

Expand the query decomposition and query processing algorithms implemented in Phase IIIA. This will include implementation of an ANSWER/RDA interface. The ANSWER query formulator produces SQL expressions, which will be submitted using an RDA-like protocol to DBMSs for execution in a distributed processing system.

5.1.3 Task IIIB.3—AI Techniques

We will investigate approaches to treatment of inconsistent data and active database issues that are required in the Army environment. We will implement selected approaches as part of the query processing algorithms.

5.1.4 Task IIIB.4—Security Study

We will develop an overall approach to security issues within ANSWER and make recommendations for exploration of key concepts. We will also explore issues associated with fault tolerance of the ANSWER system and the RDA query processing environment. Finally we will discuss the relationship of ANSWER to the ISM (Installation Support Modules).

5.1.5 Task IIIB.5—Demonstration and Training

The ANSWER tools will be demonstrated against a set of sample relational schemas representing the Installation-Level Acquisition Products and Services (ILAPS) environment based on information obtained from PM-ISM. The ANSWER/RDA interface developed in Phase IIIB.2 and the implementation of distributed query processing developed in Phase IIIB.6 will be demonstrated along with revisions to the query processing algorithms revised under Phase IIIB.3. We will provide training in the use of the software to a group selected by AIRMICS.

5.1.6 Phase IIIB.6—Distributed Query Processing

We will design and do a limited implementation of local RDA processors for two target systems and design and implement local system query translators.

5.1.7 Phase IIIB Deliverables

- Software prototype for query formulation (RDA/ANSWER integration).
- Design and implementation of integration of browser/UIMS with target demonstration platform.
- Design and implementation of query formulation back end facility (local translator design and development).
- Design integration of ANSWER tool set with target distributed environment (RDA/local system interface design and implementation).
- Oral presentation, documentation, training and demonstration (includes final report on ANSWER/ISM relationships, fault tolerance, security, and ANSWER/RDA architecture design and implementation).

5.2 Phase IV

Duration: March 1, 1991 - August 30, 1991

5.2.1 Task IV.1—Distributed Query Processing

This task will complete the integration of ANSWER tools with a distributed query processing facility. The distributed query processing facility will be the AIRMICS test-bed. The following subtasks will be performed in Phase IV.

Subtask IV.1.1—Scaling Tests: Develop new tests to explore the scalability of the solution implemented in Phase IIIB. These tests will include a larger set of query types, larger schemas, and more interaction between systems requiring some transaction session management and error handling. Execute tests on AIRMICS test-bed.

Subtask IV.1.2—Develop Local Database Language Translator Extensions: Design and implement portions of the RDA transaction processing application context as a well as new additions to the local database query language translators.

Subtask IV.1.3—Error Handling: Design and develop selected error handling facilities.

Subtask IV.1.4—Distributed System Manual: Write a manual on the steps necessary to add a new system to the RDA-based ANSWER distributed query processing network.

Subtask IV.1.5—Reports: Written monthly reports, final report, and software documentation.

5.2.2 Task IV.2—Demonstration and Training

Demonstrate the software developed in Phase IV and conduct training in the installation maintenance and use of the ANSWER system.

5.2.3 Phase IV Deliverables

- ANSWER tool set integrated with a distributed query processor running at an Army location on Army hardware .
- Documentation of software implemented in Phase IV.
- Interim and final demonstrations.
- One to two demonstrations of ANSWER software to target groups identified by AIRMICS and one training session on the installation of Phase IV software.
- Monthly reports.
- Final report.

Section 6
ANSWER Implementation

The following is a list of major files comprising the query formulator and query processing algorithms implemented and delivered for Phase IIIA.

6.1 Query Processor

The term decomposition query processor comprises the following major files:

- *query_form.pc*—This file contains the major functions that process dynamic SQL statements. The dynamic SQL statements are created for execution against individual databases after query mapping. The major functions included in this file are:

  - **UserLogin**—Manages login to Oracle.

  - **ExecuteSQL**—Contains the dynamic SQL statements required to manage execution of an arbitrary SQL command in Oracle.

  - **Warning**—Traps Oracle SQL warning messages and returns them.

- *query2.pc*—This file contains the main query decomposition and recomposition functions. Main functions in this file make use of functions from the other files. The major functions here include:

  - **SQLQuery2**—the main driver function that calls the parsing, mapping and SQL execution manager functions.

  - **SQLParser**—The main driver for parsing SQL queries into select, from and where clauses so that each clause may be mapped to individual schemas

  - **Mapping**—The main driver for mapping SQL queries from integrated schema expressions to individual schema expressions.

- *parse.c*—This file contains the major functions for parsing SQL queries including:

  - **SelectClause**— Parses the select clause from an SQL input query.
  - **FromClause**—Parses the from clause in an SQL input query.
  - **WhereClause**—Parses the where clause in an SQL query.

- *mapclauses.c*—This file contains functions pertaining to the mapping of queries from integrated schema terms to individual schema terms. It includes the following major functions:

- **MapSelectClause**—Maps the Select clause. Currently the Select clause may contain no aggregate functions. The function returns a list of attributes in the local schema. Sets global flag CompleteSelect to 1 if does NOT map completely.

- **MapFromClause**—Maps the From clause. A From clause may include more than one relation name. The mapped SQL statement for a local query is created. The function returns 1 if the query does NOT completely map to the current individual schema.

- **MapWhereClause**—Maps a Where clause. A Where clause may contain boolean tests only on the attributes of the Select clause. No Select statements embedded in the Where clause may be mapped. The function returns:

  0 if completely maps
  1 if does not completely map
  2 if there is no Where clause at all
  3 if doesn't map any (Where removed).

- **CheckForeignKey**—This function includes any foreign key conditions which are required to effectively map from an integrated schema expression to a local schema expression. The foreign key conditions are embedded in the mapping rules. The function returns the string of foreign key conditions to be added to the Where clause. If there are none, the function returns NULL.

- **CheckGlobalForeignKey**—This function adds any global join conditions that are required to effectively map from an integrated schema expression to a final recomposed schema. In cases where the query results of local schemas must be joined together to obtain final results global foreign keys are required. The global foreign key information is embedded in the mapping rules. The function returns the string of foreign key conditions to be added to the final Where clause. If there are none, the function returns NULL.

- **readrules.c**—This file contains functions to read mapping rules required for mapping integrated schema queries to individual schema queries. The following functions are defined in this file: ReadRules, ReadAttributeRules, ReadRelationRules, ReadJoinRules. The rules are currently stored in a file.

6.1.1 Limitations of Current QF Term Substitution Implementation

The current query formulator implementation has the following limitations:

1. Only conjunctive queries can be processed. This means that no queries with embedded select statements may be processed. In general queries may be transformed to conjunctive form. Additional functions would need to be written to either transform a query before processing or to extend the functionality of the existing algorithms to appropriately treat nonconjunctive queries.

2. All query communication is currently done by using files. The communication should be modified so that files are not used as intermediaries to transmit the query or query results.

6.2 Approximate Query Processing Algorithm

The approximate query processing algorithm implements a version of the Penn algorithm [BUDW89]. The approximate query processing implementation makes use of some facilities already discussed for communicating with Oracle and managing query execution and result return. This subsection will document the central functions used by approximate query processing to implement the algorithm. The main files are:

- **structures.cl**—Defines the major data structures used by the approximate query processing algorithm. Some of the functions used to execute approximate query processing are defined here as well. The functions and structures include:

  - **Rule**—A rule has a header and a left-hand side and a right-hand side. The header contains prime terms or IS terms or combination of these term; (the only supported operator includes JOIN) The left-hand side as well as right-hand side contain SDE terms or IS attributes. Some of these attributes may be identifiers in a relation. (Each side of a rule should have at least one identifier). When the header has prime terms on the right-hand side, then the left-hand side can only contain SDE terms. Similarly when the header contains IS terms, the right-hand side and the left-hand side have only integrated schema attributes. The rules may support 'Don't care' type of attributes, which is a placeholder for an arbitrary number of attributes. All are indexed by SDE or IS attributes. (Hashed by the identifier)

  - **qp-query**—Defines the query information used by the approximate query processing algorithm. This includes (1) information about standard data elements and prime terms used in the query, (2) a query result term, which represents all the attributes that must be referenced in the final result of query evaluation, and (3) the specific rules that are required to map the input query to queries against local databases.

  - **QP**—Structure that stores rules that map from integrated schema terms to local schema terms and rules that map prime terms and standard data elements to local schema terms.

  - **start-qp**—This function manages execution of the approximate query processing algorithm. It coordinates the identification of relevant rules for a given query, the construction of a query-result term, separation of the terms and the terms and the join processing required for LHS terms, and the union processing required for integrated schema terms appearing in rules as RHS terms.

- **definite-answers**—This function computes the set of tuples which qualify as definition answers to a query (essentially the join of LHS terms in the relevant rules with additional information supplied from the possible-answers).

- **possible-answers**—This function computes the set of identifiers of tuples which qualify as possible answers to a query (essentially the approximate-union of RHS terms).

- **anomalies**—This function computes the set of tuples which occur as the result of joining all the terms of relevant rules but which do not occur in the set of possible answers.

* **trw.cl**—This file contains functions for requesting creation and dropping of tables within ORACLE and functions necessary to read tables into the LISP work area from Oracle relations and to write tables out to Oracle as relations. This is necessary to execute some of the approximate query processing functions directly in ORACLE as SQL queries. This file also contains functions for creating a window to present the results of query execution in a scrollable window.

* **penn-union.cl**—This file contains functions necessary to perform the approximate union and promotion operations.

* **qparser.cl**—This file contains functions that parse the query received as input from the query formulator into internal structures required for the approximate union algorithm

6.2.1 Limitation of Approximate Query Processing

The current algorithm is well defined only for cases where all relations being manipulated by the algorithm share the same key attribute. This limits the applicability of the algorithm.

6.3 Query Formulator

The query formulator provides assistance in the creation and editing of SQL queries using text or graphic input. The major query formulator functions are in the following files:

* **core.c**—This file contains functions that implement the main functions of the query formulator such as editing functions (e.g., delete-item, add-item), and buffer management functions.

* **sqlparser.cl**—This file contains functions that parse SQL text supplied as input to the query formulator. The parsed queries are stored in internal data structures used by the query formulator to support editing functions. The parser is designed to tolerate ill-formed SQL and to supply place-holders for

portions of a query that cannot be parsed as correct SQL. This allows the user to create partially formed SQL queries and still make effective use of the SQL editing functions.

- **sqlprinter.cl**—This file provides functions for managing the printing of complete or partial SQL statements in the query formulator editing windows. The SQL statements may be user-typed, insertions from a previously saved query, or portions of a query developed graphically in the condition expression editor. The sqlprinter functions must correctly format the SQL statements for printing within the editing window.

- **modes.cl**—This file contains functions that define the specific editing modes supported by the query formulator. The modes are based on the items being edited and include such things as: condition, expression, select clause, Boolean operator, and others. The modes are used to manage the editing actions available for the current item.

- **structs.cl**—This file contains functions that define the major data structures used by the query formulator. The structures defined include; query-buffer, select-clause, condition clauses, leaf and non-leaf nodes in a condition tree, operator and others.

- **operators.cl**—This file contains functions that define the specific SQL operators available to be used as part of an SQL query within the condition expression editor. This set of operators may be extended in the future to expand the coverage of editable statements available in the SQL Where clause.

- **graph.cl**—This file contains functions that define the query formulator main window, the commands buttons global to the query formulator and the pop-up windows used with the left-hand window in the query formulator.

- **qwindow.cl**—This file contains functions that define the window management functions for the condition expression editor (the right window) in the query formulator.

- **term.cl**—This file contains window management functions for the term window.

- **script.cl**—This file contains the functions that define the state transition representation of the query formulator used by the ANSWER user interface manager.

Section 7
References

[AR 25-9] Army Regulation 25-9, Army Data Management and Standards Program, 1989.

[CERI 83] S. Ceri, G. Pelagatti, "Correctness of Query Execution Strategies in Distributed Databases," *ACM Transactions on Database Systems*, Vol. 8, No. 4, December 1983, pp. 577-607.

[BUDW 89] O.P. Buneman, S.B. Davidson, A. Walters, "Federated Approximations for Heterogeneous Databases," *IEEE Technical Committee on Office Automation Newsletter*, Vol. 3, No. 2, August 1989, pp. 27-34.

[CHUN 90] C-W. Chung, "Dataplex: An Access to Heterogeneous Distributed Databases," *Communications of the ACM*, Vol. 33, No. 1, 1990, pp. 70-80.

[DAYA 84] U. Dayal, H-Y. Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, November 1984, pp. 628-645.